**Morgun Leonard**

**CS-320 Software Test, Automation**

**7-2 Project Two**

**Summary and Reflections Report**

**Unit Testing Approach**

For Project One, I implemented unit tests using JUnit to validate the functionality of the Contact, Task, and Appointment services. Each service was tested independently to ensure compliance with software requirements. My approach focused on verifying:

* Correct data validation (e.g., field length restrictions, null values, and invalid inputs)
* CRUD operations (Create, Read, Update, Delete)
* Error handling and exception management
* Edge cases testing (boundary conditions and unexpected input handling)

I aligned this approach with software requirements by ensuring that all business rules were enforced. For example, appointment dates could not be set in the past, phone numbers required exactly ten digits, and contact names had a maximum of ten characters. The success of these tests was confirmed by asserting expected outcomes and handling edge cases effectively.

**Effectiveness of JUnit Tests**

The effectiveness of the tests was measured using test coverage analysis. The goal was to achieve at least 80% test coverage across all Java classes. My tests verified multiple aspects of functionality, including:

* Positive test cases: Verifying expected behaviors when correct input is provided.
* Negative test cases: Ensuring proper exception handling for invalid data.
* Boundary testing: Checking limits, such as string length constraints and date validation.
* Performance impact analysis: Ensuring the tests did not introduce unnecessary execution overhead.

By analyzing the test results, I confirmed that all major functions were covered, and no critical paths were left untested. The use of an automated coverage tool allowed me to detect areas that needed additional test cases, ensuring a robust verification process.

**Experience Writing JUnit Tests**

Writing the JUnit tests required careful consideration of testing strategies to ensure technically sound and efficient code. Strategies included:

* Isolation testing: Each test focused on a single function without dependencies.
* Parameterized tests: Some tests used different input values to verify robustness.
* Assertions: Used to confirm expected behaviors (e.g., assertEquals(), assertThrows()).
* Test-driven development (TDD): Writing test cases before implementing functionality to drive high-quality design decisions.

For example, in ContactServiceTest.java, I used the following assertion to verify that a duplicate contact could not be added:

assertThrows(IllegalArgumentException.class, () -> service.addContact(contact));

To ensure efficiency, redundant test cases were avoided, and common setup operations were grouped using @BeforeEach methods. This improved maintainability and reduced repetitive code. The test execution time was minimized by optimizing test conditions and utilizing mock objects where applicable.

**Reflection**

**Testing Techniques Used**

The primary software testing techniques employed in this project included:

* Unit Testing: Focused on validating individual components in isolation.
* Boundary Testing: Ensured constraints (e.g., max character limits) were enforced.
* Exception Testing: Verified proper handling of invalid input scenarios.
* Regression Testing: Ensured that modifications did not introduce new defects.

Other testing techniques that were not used but could be applicable in different projects include:

* Integration Testing: Verifies interactions between multiple components.
* System Testing: Ensures the application functions correctly as a whole.
* Performance Testing: Measures speed and responsiveness under load.
* Security Testing: Ensures protection against vulnerabilities such as SQL injection and data leaks.

These alternative techniques are more relevant for full-system validation rather than unit-level testing. However, integrating them into the development lifecycle can greatly enhance software robustness.

**Practical Uses and Implications**

Each testing technique has specific applications:

* Unit tests help in identifying defects early and reducing debugging time.
* Integration tests validate seamless data flow between different system modules.
* System tests ensure that the software meets end-user expectations.
* Performance tests help in identifying bottlenecks and optimizing system efficiency.
* Security tests ensure applications are resilient against cyber threats.

For example, in a banking application, integration testing would be crucial to ensure that account transactions and balance updates function correctly across different services. In a web-based application, performance testing would be necessary to ensure scalability under high user loads.

**Mindset and Caution in Testing**

Throughout this project, I employed a cautious and detail-oriented approach to testing. It was critical to appreciate the complexity of code interactions and prevent silent failures that could lead to system issues.

For instance, while testing the Appointment class, I ensured that past dates were strictly rejected to avoid incorrect scheduling:

assertThrows(IllegalArgumentException.class, () -> new Appointment(pastDate, "Meeting"));

This demonstrates the importance of anticipating edge cases and thoroughly verifying constraints. Additionally, I utilized log analysis to verify error-handling mechanisms worked correctly under various conditions.

**Limiting Bias in Testing**

Bias in testing can occur if a developer assumes their code is correct. To reduce this risk, I:

* Wrote tests before writing implementation code (Test-Driven Development mindset).
* Included negative tests to ensure expected failures were handled.
* Reviewed test cases critically to cover potential blind spots.
* Used third-party static analysis tools to detect potential issues before runtime.

If I were responsible for testing my own code in a real-world project, I would involve peer reviews and automated testing tools to detect issues I might overlook. Additionally, I would implement randomized input generation to further uncover edge cases.

**Discipline and Avoiding Technical Debt**

Maintaining discipline in software testing ensures long-term code quality. Cutting corners in testing can lead to undiscovered bugs, higher maintenance costs, and unreliable software.

For example, skipping validation tests might result in user data corruption or security vulnerabilities. To prevent this, I plan to:

* Follow structured testing frameworks (like JUnit)
* Use automated test coverage tools to track untested code
* Document test cases thoroughly for future reference
* Continuously refactor and optimize tests to avoid legacy test bloat

By committing to rigorous testing, I aim to develop reliable, maintainable, and scalable software solutions.

**Conclusion**

This project reinforced the importance of structured unit testing in software development. By following a methodical approach, implementing effective tests, and reflecting on software testing best practices, I ensured that the system met the requirements and maintained high-quality standards. Moving forward, I will continue refining my testing strategies to improve software quality and efficiency.